Neural Networks

Agenda

References

A few classical/useful monographs:

- **Deep Learning** by Goodfellow, Bengio and Courville (2015).
- **Neural networks and statistical learning** by Du and Swamy (2013)
- **Deep Learning with Python/R** by Chollet (/Allaire) (2018-2022)
- **Understanding Deep Learning** by Prince (2024).

Caution: the term neural network is misleading, it has little to do with the human brain (of which we know relatively little). F. Chollet defined NN as

"*chains of differentiable, parameterised geometric functions, trained with gradient descent (with gradients obtained via the chain rule)*".

History

Some milestones

- 1957: first **perceptron** by Rosenblatt
- 1986: first applications of **backpropagation** for multilayer perceptrons
- 1997: invention of LSTM, a popular recurrent NN
- acceleration and diversification since 2012 (AlexNet at ImageNet divided the error rate by 2, from 30% to 15%!): use of CNN for computer vision, RNN for translation, AlphaGo/AlphaZero/AlphaStar via RL.
- transformers in 2017: $\rightarrow$ chatGPT in 2022, GPT-4. Sky is the limit!

$\rightarrow$ the basic ideas are pretty **old**!
NN have been considered since the 1990s for financial prediction (mostly by **computer scientists**), but it's only since the 2010s that they have reached their full potential (still developing).

Notations

Valid for the whole deck.

- The data is separated into a matrix $\boldsymbol{X} = x_{i,j}$ of features and a vector of output values $\boldsymbol{y} = y_i$. $\boldsymbol{x}$ or $\boldsymbol{x}_i$ denotes one line of $\boldsymbol{X}$.
- A neural network will have L layers.
- For each layer l, the number of units is U_l.
- The weights for unit k located in layer l are denoted with $\boldsymbol{w}_k^{(l)} = w_{k,j}^{(l)}$ and the corresponding biases $b_k^{(l)}$. The length of $\boldsymbol{w}_k^{(l)}$ is equal to U_{l-1}. k refers to the location of the unit in layer l while j to the unit in layer $l-1$.
- Outputs (post activation) are denoted $o_{i,k}^{(l)}$ for instance/occurrence i, layer l and unit k.
- All vectors are column vectors, $\boldsymbol{v}'$ is the transpose of $\boldsymbol{v}$.

Agenda

The simple building block

The original aim of the perceptron is **binary classification**. Let's say in our case that the output is $\{0 = \text{do not invest}\}$ versus $\{1 = \text{invest}\}$ (e.g., derived from return, negative vs positive).

Activated linear mapping!

The model is the following:

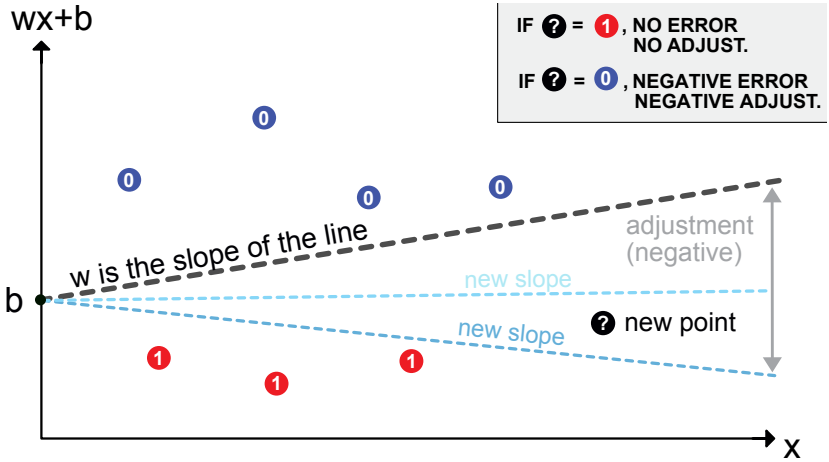
$$f(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{x}'\boldsymbol{w} + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

The vector of weights $\boldsymbol{w}$ scales the variables and the bias b shifts the decision barrier.

Given values for b and w_i, the error is $\epsilon_i = y_i - 1_{\left\{\sum_{j=1}^{J} x_{i,j} w_j + w_0 > 0\right\}}$.
We set $b = w_0$ and add an initial constant column to x: $x_{i,0} = 1$, so that $\epsilon_i = y_i - 1_{\left\{\sum_{j=0}^{J} x_{i,j} w_j\right\}}$.

Training (illustration)

How to minimise the error? $\neq$ regression: no closed form solution!



Training (in practice)

Each (training) data point participates to the determination of the w_i.

The updating algorithm (learning) is the following

for each pair of data $(y_i, \boldsymbol{x}_i)$,

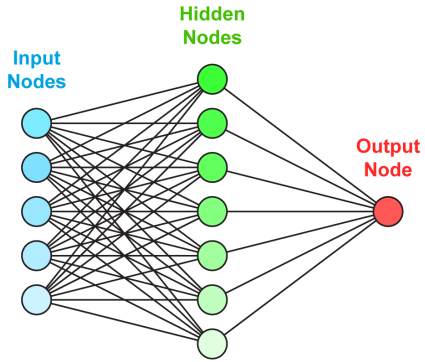
1. compute the current model value at point $\boldsymbol{x}_i$: $\tilde{y}_i = 1_{\left\{\sum_{j=0}^{J} w_j x_{i,j} > 0\right\}}$,
2. adjust the weight: $w_j \leftarrow w_j + \eta(y_i - \tilde{y}_i)x_{i,j}$,

where $\eta > 0$ is the **learning rate**: a small η will only marginally alter the weights, but a large η might move them too far. Usually, $\eta < 1$. When η is very small, convergence is slow but almost sure; when it is large, convergence is faster but unsure.

Agenda

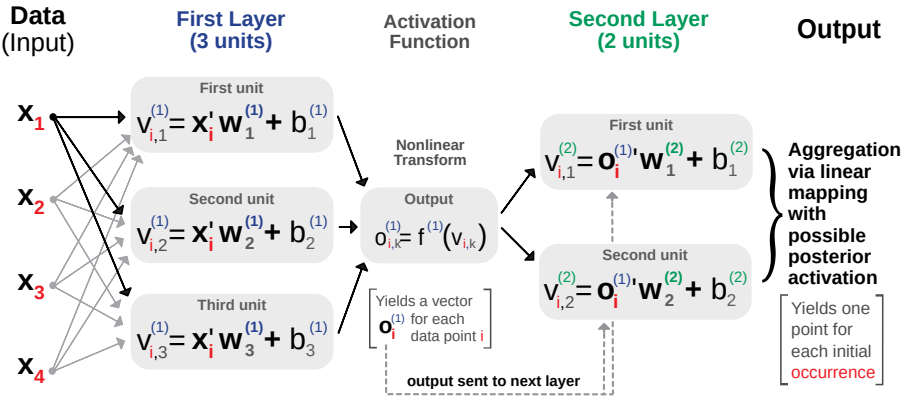
Combinations of perceptrons

Usual representation (source: butleranalytics):



This hides what is really happening: there is a perceptron inside the green circles!
The green phase is a layer, and there can be many $\rightarrow$ deep network.

The complete scheme (example)



Notation:

Weights

$w^{(l)}_k$ — (l) index of layer; k index of unit

Intermed. Values

$v^{(l)}_{i,k}$ — (l) index of layer; i index of unit; k index of occurrence

Activation functions are usually indexed according to the layer they succeed

A formal definition

Here it goes (case with simple real output)

values:

$$v_{i,k}^{(1)} = \boldsymbol{x}_i' \boldsymbol{w}_k^{(1)} + b_k^{(1)}, \qquad \text{for } l = 1, \quad k \in [1, U_1]$$

$$v_{i,k}^{(l)} = (\boldsymbol{o}_i^{(l-1)})' \boldsymbol{w}_k^{(l)} + b_k^{(l)}, \text{for } l \geq 2, \quad k \in [1, U_l]$$

output:

$$o_{i,k}^{(l)} = f^{(l)} \left(v_{i,k}^{(l)} \right)$$

terminal:

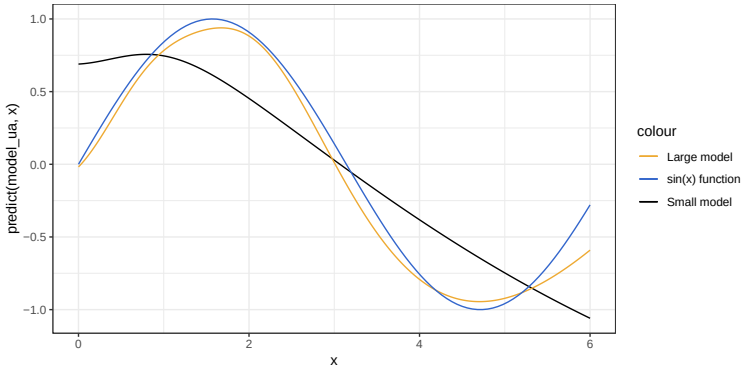
$$\tilde{y}_i = f^{(L+1)} \left((\boldsymbol{o}_i^{(L)})' \boldsymbol{w}^{(L+1)} + b^{(L+1)} \right)$$

i is data row number, l is layer number, $k \in [1, U_l]$ is unit number.
The last activation function can simply be the identity function $f^{(L+1)}(x) = x$.

Fundamental property

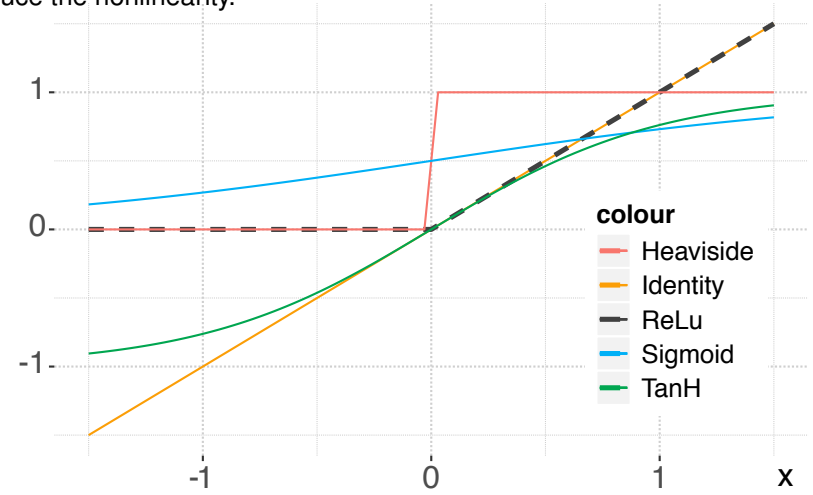
Universal approximation

Any continuous function f can be **approximated up to arbitrary precision** (on a finite interval) by a single layer perceptron with smooth (bounded continuous) activation function. How come? You just need to add units to help the perceptron overfit!



Activation functions

They induce the nonlinearity.



Backpropagation (1/6)

A idea simple in theory

The aim is to minimise the total error / distance / discrepancy

$$E = \sum_{i=1}^{N} d(y_i, \tilde{y}_i(\boldsymbol{W})) := \sum_{i=1}^{N} D(\tilde{y}_i),$$

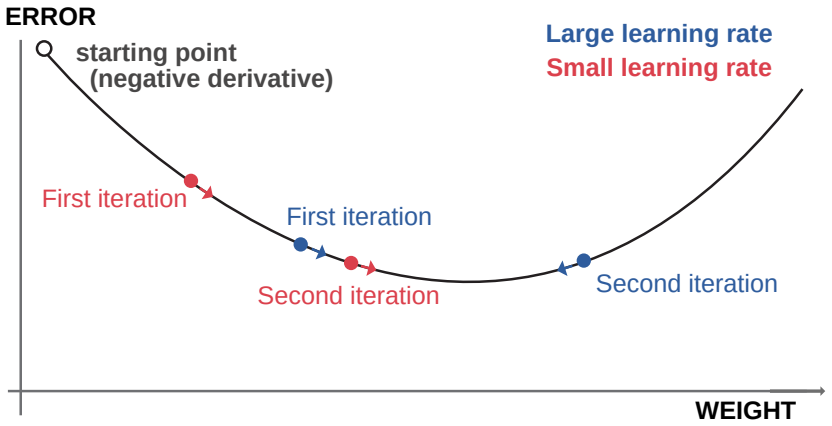
where $\boldsymbol{W}$ encompasses all parameters, i.e., all **weights** and all **biases** of all units of all layers (in computer vision, there are millions of parameters). Each term of the sum can be minimised separately through gradient descent. The updating of weights is done through

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial D(\tilde{y}_i)}{\partial \boldsymbol{W}}$$

The computation of $\frac{\partial D(\tilde{y}_i)}{\partial \boldsymbol{W}}$ is a real challenge because of all the layers! Note, we will often work with $D(x) = (x - y_i)^2$ (quadratic error, the classical choice for regression analysis).

Gradient Descent

In one illustration, in dimension 1:



Backpropagation (2/6)

We recall $v_{i,k}^{(l)} = (\boldsymbol{o}_i^{(l-1)})' \boldsymbol{w}_k^{(l)} + b_k^{(l)} = b_k^{(l)} + \sum_{j=1}^{U_l} o_{i,j}^{(l-1)} w_{k,j}^{(l)}$.

Parameters

For each layer and each unit k, we must '**estimate**' (find, optimise) the best possible values for $\boldsymbol{w}_k^{(l)}$ and $b_k^{(l)}$.

- For the first layer, this gives $(U_0 + 1)U_1$ parameters, where U_0 is the number of columns in $\boldsymbol{X}$ (i.e., number of explanatory variables)
- For layer $l \in [2, L]$, the number of parameters is $(U_{l-1} + 1)U_l$
- For the final output, there are simply $U_L + 1$ parameters
- In total, this means the total number of values to optimise is

$$\mathcal{N} = \left(\sum_{l=1}^{L} (U_{l-1} + 1) U_l \right) + U_L + 1$$

Backpropagation (3/6)

We recall:
$\tilde{y}_i = f^{(L+1)}\left((\boldsymbol{o}_i^{(L)})' \boldsymbol{w}^{(L+1)} + b^{(L+1)}\right) = f^{(L+1)}\left(b^{(L+1)} + \sum_{k=1}^{U_L} w_k^{(L+1)} o_{i,k}^{(L)}\right)$

First step: last weights and bias

$$\begin{aligned}
\frac{\partial D(\tilde{y}_i)}{\partial w_k^{(L+1)}} &= D'(\tilde{y}_i)\left(f^{(L+1)}\right)'\left(b^{(L+1)} + \sum_{k=1}^{U_L} w_k^{(L+1)} o_{i,k}^{(L)}\right) o_{i,k}^{(L)} \\
&= D'(\tilde{y}_i)\left(f^{(L+1)}\right)'\left(v_{i,k}^{(L+1)}\right) o_{i,k}^{(L)} \\
\frac{\partial D(\tilde{y}_i)}{\partial b^{(L+1)}} &= D'(\tilde{y}_i)\left(f^{(L+1)}\right)'\left(b^{(L+1)} + \sum_{k=1}^{U_L} w_k^{(L+1)} o_{i,k}^{(L)}\right)
\end{aligned}$$

Henceforth, we leave the case of the bias aside, as it can be considered as a particular case of a weight.

Backpropagation (4/6)

We recall $v_{i,k}^{(l)} = (\boldsymbol{o}_i^{(l-1)})' \boldsymbol{w}_k^{(l)} + b_k^{(l)} = b_k^{(l)} + \sum_{j=1}^{U_l} o_{i,j}^{(l-1)} w_{k,j}^{(l)}$.

Propagation through the chain rule

For one particular weight in layer L, the impact is channelled through all outputs $\boldsymbol{o}_k^{(L)}$. The chain rule (composition of function) yields

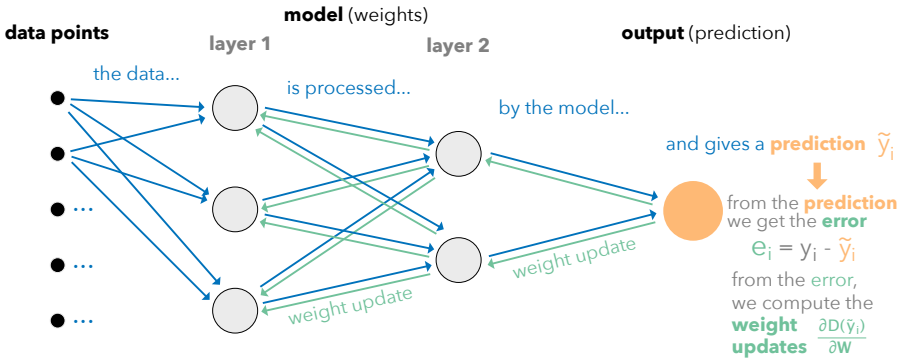
$$\begin{aligned}
\frac{\partial D(\tilde{y}_i)}{\partial w_{k,j}^{(L)}} &= \frac{\partial D(\tilde{y}_i)}{\partial v_{i,k}^{(L)}} \frac{\partial v_{i,k}^{(L)}}{\partial w_{k,j}^{(L)}} = \frac{\partial D(\tilde{y}_i)}{\partial v_{i,k}^{(L)}} o_{i,j}^{(L-1)} \\
&= \frac{\partial D(\tilde{y}_i)}{\partial o_{i,k}^{(L)}} \frac{\partial o_{i,k}^{(L)}}{\partial v_{i,k}^{(L)}} o_{i,j}^{(L-1)} = \frac{\partial D(\tilde{y}_i)}{\partial o_{i,k}^{(L)}} (f^{(L)})'(v_{i,k}^{(L)}) o_{i,j}^{(L-1)} \\
&= D'(\tilde{y}_i) \left(f^{(L+1)}\right)' \left(v_{i,k}^{(L+1)}\right) w_k^{(L+1)} (f^{(L)})'(v_{i,k}^{(L)}) o_{i,j}^{(L-1)}
\end{aligned}$$

To access $D(\tilde{y}_i)$ from $w_{k,j}^{(L)}$,

$$w_{k,j}^{(L)} \leftarrow v_{i,k}^{(L)} \leftarrow o_{i,k}^{(L)} \text{ (i.e., } f^{(L)}) \leftarrow D(\tilde{y}_i)$$

Backpropagation (5/6)

- The process continues iteratively (i.e., **backwards** until the beginning of the network). Luckily, there is a simple 'form' or 'trick' to derive the derivatives in a greedy manner.
- The idea is to **recycle** the computation from layer l for ayer $l-1$ (there is only one new term to calculate for each new node).
- We recommend Section 4.3 from Du and Swamy (2013) for more details on the topic.



Backpropagation (6/6)

Back to $E = \sum_{i=1}^{N} D(\tilde{y}_i)$. How often are the weights updated?

- **Batch update**: take the full training sample (needs memory, rare updates);
- **SGD** (stoch. grad. desc): update one instance at a time (lots of noisy updates),
- **mini-batch**: in between: use blocks of data. Best of two worlds?

Terminology

- **batch size**: number of samples used to update the weights. Each sample has one forward and backward pass with the 'old' weights. After the whole batch has passed, the weights are updated
- **epoch**: one forward and backward pass of the whole training sample
- **iteration**: the number of passes. Sometimes, it's the number of passes required to achieve on epoch, sometimes, it's the total number of passes (i.e., the previous one multiplied by the number of epochs).

Total updates = (nb epochs) * N / (batch size).
Usually, for large samples, the batch size is between 20 and 2000 (often in powers of 2, like 512 and 1024).
The user will be asked to specify both the **batch size** and the **number of epochs**.

Classification

Obviously, the problem is different.

Multidimensional output

When the output is a **nominal category**, the output for one occurrence is not one single number, but a vector (usually stemming from **one-hot encoding**). E.g.:

red	1	0	0
green	0	1	0
blue	0	0	1

If the output colour is green, the the vector is (0,1,0). Hence, the output (final layer) of the network must also be a vector. Usually, it's a vector of numbers that sum to one which indicate the 'probability' of the occurrence to belong to each category. Like (0.1,0.8,0.1) would be an indication towards green. One typical activation function in this case is the **softmax** function:

$$s(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}},$$

where $\boldsymbol{x}$ is K-dimensional and $s(\boldsymbol{x})_i$ is the i^{th} element of the output vector.

Deep learning?

How deep should we go?
In Finance, not too deep! (3-4 hidden layers at most)

Network size

First, the so-called **geometric pyramid rule**: the number of units decreases geometrically as the network advances. If there are L hidden layers, with I features in the input and O dimensions in the output (often, $O = 1$), then, for the l^{th} layer, a rule of thumb for the number of units is

$$U_l = O\left(\frac{I}{O}\right)^{\frac{L+1-l}{L+1}}$$

If there is only one intermediate layer, the recommended proxy is $\sqrt{IO}$.

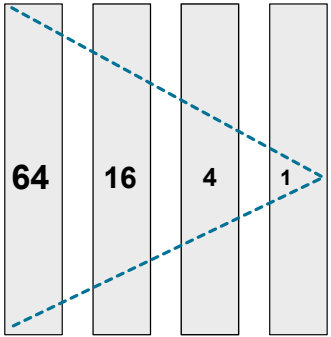
Second, sadly, there is no rule that sets the number of parameters (weights+biases) depending on the number of **features** and **observations**. It is recommended that $\mathcal{N}$ be smaller than a small fraction of the number of observations (usually 20% or 10% at most).

Simple rule

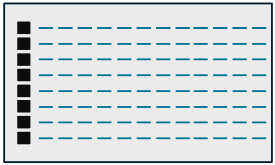
(simple slide)

The geometric rule

(example)



Sample / parameter ratio



N = sample size = # rows

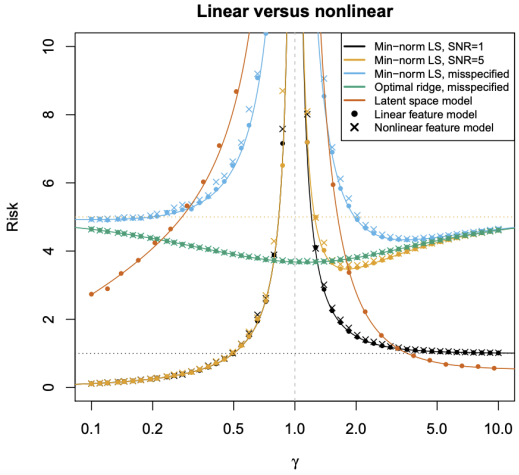
M = # parameters (biases+weights)

N/M >> 10

Double descent

Recently, results have show that complexity may work well!

- Benign Overfitting in Linear Regression
- Surprises in High-Dimensional Ridgeless Least Squares Interpolation



Agenda

1. Introduction
2. The perceptron
3. Multilayer perceptron (MLP)
4. Recurrent networks
5. In practice
6. Wrap-up

Principle

By definition, MLP:

- are feed-forward (one direction)
- have no memory

In some cases with **sequential linkages** (e.g., time-series or speech recognition), it might be useful to keep track of what happened with the previous sample (i.e., there is a natural ordering).

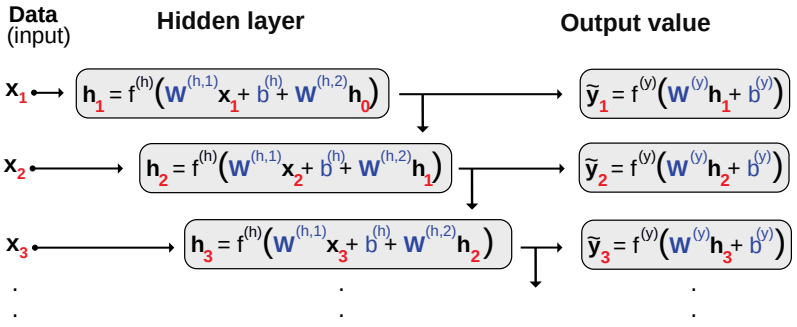
In short, with one layer

$$\tilde{y}_i = f^{(2)}\left(\sum_{j=1}^{U_1} o_{i,j} w_j^{(2)} + b^{(2)}\right)$$

$$o_{i,j} = f^{(1)}\left(\sum_{k=1}^{U_0} x_{i,k} w_k^{(1)} + b^{(1)} + \sum_{k=1}^{U_1} W_k^{(h)} o_{i-1,k}\right)$$

RNN can be viewed/unfolded as high dimensional feed-forward networks.
Problem: **vanishing gradients**. The derivative values at the node in the backprop are smaller than one in absolute value (e.g., tanh). Iterative (BPTT) multiplications imply adjustments close to zero in the early layers.

Unfolding Recurrent networks



The first instance impacts all the network; the last instance impacts only the last layer.
Problem: the successive iterations of the activation function yield vanishing gradients.

Gated Recurrent Units (GRU)

(see also Long-Short Term Memory, LSTM for another popular RNN) The processes add '**memory gates**' at several stages.

The definition (see Cho et al. (2014))

$$\tilde{y}_i = z_i \tilde{y}_{i-1} + (1 - z_i) \tanh\left(\boldsymbol{w}_y' \boldsymbol{x}_i + b_y + u_y r_i \tilde{y}_{i-1}\right) \quad \text{output}$$
$$z_i = \text{sig}(\boldsymbol{w}_z' \boldsymbol{x}_i + b_z + u_z \tilde{y}_{i-1}) \quad \text{'update gate'} \in (0,1)$$
$$r_i = \text{sig}(\boldsymbol{w}_r' \boldsymbol{x}_i + b_r + u_r \tilde{y}_{i-1}) \quad \text{'reset gate'} \in (0,1)$$

$$\tilde{y}_i = \underbrace{z_i}_{\text{weight}} \underbrace{\tilde{y}_{i-1}}_{\text{past value}} + \underbrace{(1 - z_i)}_{\text{weight}} \underbrace{\tanh\left(\boldsymbol{w}_y' \boldsymbol{x}_i + b_y + u_y r_i \tilde{y}_{i-1}\right)}_{\text{candidate value (classical RNN)}}$$

z_i: current and past values decide the optimal mix between the two.
r_i: for the candidate value, r_i decides which amount of past to retain.
$\rightarrow$ Because of dimension constraints, training of these models is not simple!

Other NN architectures

The family of NNs is diverse!

- **Convolutional Neural Networks**: mostly used for computer vision because they progressively reduce the dimension of the input (image = rectangle of pixels = matrices of numbers between 0 and 255 - one layer for RGB).
- **Autoencoders**: the input is transformed through the network but the purpose is to find an output as close as possible to the input (i.e., input = output!). This helps find a simpler representation of the input (nonlinear PCA).
- **Generative Adversarial Networks**: two networks compete. The first one seeks to optimize a loss function, but the second one is tweaking the data to make it harder for the first one to reach its goal.
- **Tabular Networks**: NNs for tabular data!
- **Transformers**: for NLP (and more?)

Agenda

Keras (1/4) !!!

Great high-level API. See

- https://keras.io for the original Python implementation
- https://keras3.posit.co/ for the R version

In four steps

1. architecture definition
2. loss function & optimisation
3. the fit/train line
4. prediction

Keras (2/4)

Architecture

We will use 2 types of **layers**: dense (classical) and GRU. The inputs/arguments are (don't copy paste):

1. **layer_dense**(units = 32,
   activation = 'relu',
   input_shape = c(ncol(train_data)))
2. **layer_gru**(units = 32,
   activation = 'tanh',
   recurrent_activation = 'hard_sigmoid',
   use_bias = TRUE)

- All arguments are straightforward: they correspond to the concepts seen above.
- The input shape is only required for the first layer.
- It is possible to specify the initialisation of weights.
- A list of activation functions can be found: activation-layers

Keras (3/4)

Loss functions and optimisation

The syntax is easy:

```
model %>% compile(
  loss = 'mean_absolute_error',
  optimizer = optimizer_rmsprop(),
  metrics = c('mean_absolute_error')
)
```

- All arguments are again straightforward.
- Other **optimisers** and **loss functions** are possible (see Optimizers and Losses in Keras Reference).

Keras (4/4)

Finally,

```
NN <- model %>% fit(
   training_features,
   training_label,
   epochs = 30,
   batch_size = 32,
   validation_data = list(testing_features, testing_label)
)
plot(NN) # to see convergence of loss
```

- The first two lines correspond to the training data.
- The next two inputs to training parameters.
- Finally, the last line specifies the testing sample.
- The plot show the improvement of performance w.r.t. the number of epochs

Training & **prediction** are more tricky for RNN.

Agenda

In short

Remember

- NN are **compositions of linear mappings** with **intermediate non-linear activations**
- training is done through **backpropagation** of gradient / errors
- Keras is a great tool to implement NN easily

Overfitting/**generalisation** is a major issue.

Thank you for your attention

Any questions?